Title	Set containment join revisited
Author(s)	Bouros, P; Mamoulis, N; GE, S; Terrovitis, M
Citation	Knowledge and Information Systems, 2016, v. 49, p. 375-402
Issued Date	2016
URL	http://hdl.handle.net/10722/246193
Rights	The final publication is available at Springer via http://dx.doi.org/[insert DOI]; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Set Containment Join Revisited

Panagiotis Bouros · Nikos Mamoulis · Shen Ge · Manolis Terrovitis

Abstract Given two collections of set objects R and S, the $R \bowtie_{\subseteq} S$ set containment join returns all object pairs $(r, s) \in R \times S$ such that $r \subseteq s$. Besides being a basic operator in all modern data management systems with a wide range of applications, the join can be used to evaluate complex SQL queries based on relational division and as a module of data mining algorithms. The state-of-the-art algorithm for set containment joins (`PRETTI`) builds an inverted index on the right-hand collection S and a prefix tree on the left-hand collection R that groups set objects with common prefixes and thus, avoids redundant processing. In this paper, we present a framework which improves `PRETTI` in two directions. First, we limit the prefix tree construction by proposing an adaptive methodology based on a cost model; this way, we can greatly reduce the space and time cost of the join. Second, we partition the objects of each collection based on their first contained item, assuming that the set objects are internally sorted. We show that we can process the partitions and evaluate the join while building the prefix tree and the inverted index progressively. This allows us to significantly reduce not only the join cost, but also the maximum memory requirements during the join. An experimental evaluation using both real and synthetic datasets shows that our framework outperforms `PRETTI` by a wide margin.

Keywords Set-valued data · containment join · query processing · inverted index · prefix tree

To appear at the Knowledge and Information Systems Journal (KAIS).

Panagiotis Bouros
Department of Computer Science, Aarhus University, Denmark
E-mail: pbour@cs.au.dk

N. Mamoulis · S. Ge
Department of Computer Science, The University of Hong Kong, Hong Kong SAR, China
E-mail: {nikos,sge}@cs.hku.hk

M. Terrovitis
Institute for the Management of Information Systems, Research Center "Athena", Greece
E-mail: mter@imis.athena-innovation.gr

1 Introduction

Sets are ubiquitous in computer science and most importantly in the field of data management; they model among others transactions and scientific data, click streams and Web search data, text. Contemporary data management systems allow the definition of set-valued (or multi-valued) data attributes and support operations such as containment queries [1,23,37,38,42]. Joins are also extended to include predicates on sets (containment, similarity, equality, etc.) [21]. In this paper, we focus on the efficient evaluation of an important join operator: the set containment join. Formally, let R, S be two collections of set objects, the $R \bowtie_{\subseteq} S$ set containment join returns all pairs of objects $(r, s) \in R \times S$ such that $r \subseteq s$.

Application examples/scenarios. Set containment joins find application in a wide range of domains for knowledge and data management. In decision support scenarios, the join is employed to identify resources that match a *set* of preferences or qualifications, e.g., on real estate or job agencies. Consider a recruitment agency which besides publishing job-offers also performs a first level filtering of the candidates. The agency retains a collection of job-offers R where an object r contains the set of required skills for each job, and a collection of job-seekers S with s capturing the skills of each candidate. The $R \bowtie_{\subseteq} S$ join returns all pairs of jobs and qualifying candidates for them which the agency then forwards to job-offerers for making the final decision. Containment joins can also support critical operations in data warehousing. For instance, the join can be used to compare different versions of set-valued records for entities that evolve over time (e.g., sets of products in the inventories of all departments in a company). By identifying records that subsume each other (i.e., a set containment join between two versions), the evolution of the data is monitored and possibly hidden correlations and anomalies are discovered.

In the core of traditional database systems and data engineering, set containment joins can be employed to evaluate complex SQL queries based on division [13,32]. Consider for example Figure 1 which shows two relational tables. The first table shows students and the courses they have passed, while the second table shows the required courses to be taken and passed in order for a student to acquire a skill. For example, Maria has passed Operating systems and Programming. As the courses required for a Systems Programming skill are Operating systems and Programming, it can be said that Maria has acquired this skill. Consider the query "for each student find the skills s/he has acquired" expressed in SQL below:

```
select P1.Student, R1.Skill
from Passes as P1, Requires as R1
where not exists (select R2.Course
                  from Requires as R2
                  where R1.Skill = R2.Skill
                  and not exists (select P2.Course
                                  from Passes as P2
                                  where P2.Student=P1.Student
                                  and P2.Course=R2.Course));
```

It is not hard to see that this query is in fact a set containment join between tables Requires and Passes, considering each skill and student as the set of courses they require or have passed, respectively. This example demonstrates the usefulness of set containment joins even in classic databases with relations in 1NF.

Student	Course
John	Algorithms
Peter	Databases
Maria	Op. Systems
Peter	Programming
John	Databases
Maria	Programming
Peter	Op. Systems

(a) table Passes

Skill	Course
DBA	Databases
DBWeb	Databases
DBWeb	Programming
Sys. Prog.	Programming
Sys. Prog.	Op. Systems

(b) table Requires

Fig. 1 Example of relational division based on set containment join: "for each student find the skills s/he has acquired"

In the context of data mining, containment join can act as a module during frequent itemset mining [31]. Consider the classic Apriori algorithm [2] which is well-known for its generality and adaptiveness to mining problems in most data domains; besides, studies like [43] report that Apriori can be faster than FP-growth-like algorithms for certain support threshold ranges and datasets. At each level, the Apriori algorithm (i) generates a set of candidate frequent itemsets (having specific cardinality) and (ii) counts their support in the database. Candidates verification (i.e., step (ii)), which is typically more expensive than candidates generation (i.e., step (i)), can be enhanced by applying a set containment join between the collection of candidates and the collection of database transactions. The difference is that we do not output the qualifying pairs, but instead count the number of pairs where each candidate participates (i.e., a join followed by aggregation).

Motivation. The above examples highlight not only the range of applications for set containment join but also the importance of optimizing its evaluation. Even though this operation received significant attention in the past with a number of algorithms proposed being either signature [21, 28, 29, 30] or inverted index based [24, 27], to our knowledge, since then, there have not been any new techniques that improve the state-of-the-art algorithm `PRETTI` [24]. `PRETTI` evaluates the join by employing an inverted index I_S on the right-hand collection S and a prefix tree T_R on the left-hand collection R that groups set objects with common prefixes in order to avoid redundant processing. The experiment analysis in [24] showed that `PRETTI` outperforms previous inverted index-based [27] and signature-based methods [29, 30], but as we discuss in this paper, there is still a lot of room for improvement primarily due to the following two shortcomings of `PRETTI`. First, the prefix tree can be too expensive to build and store, especially if R contains sets of high cardinality or very long. Second, `PRETTI` completely traverses the prefix tree during join evaluation, which may be unnecessary, especially if the set of remaining candidates is small.

Contributions. Initially, we tackle the aforementioned shortcomings of `PRETTI` by proposing an *adaptive* evaluation methodology. In brief, we avoid building the entire prefix tree T_R on left-hand collection R which significantly reduces the requirements in both space and indexing time. Under this *limited* prefix tree denoted by ℓT_R, the evaluation of set containment join becomes a two-phase procedure that involves (i) *candidates generation* by traversing the prefix tree, and (ii) *candidates verification*. Then, we propose a *cost model* to switch *on-the-fly* to candidates ver-

ification if the cost of verifying the remaining join candidates in current subtree is expected to be lower than prefix-tree based evaluation, i.e., candidates generation.

Next, we propose the *Order and Partition Join* (`OPJ`) paradigm which considers the items of each set object in a particular order (e.g., in decreasing order of their frequency in the objects of $R \cup S$). Collection R and S are divided into partitions such that R_i (S_i) contains all objects in R (S) for which the first item is i. Then, for each item i in order, `OPJ` processes partitions R_i and S_i by (i) updating inverted index I_S to include all objects in S_i and (ii) creating prefix tree T_{R_i} for partition R_i and joining it with I_S. As the inverted index is *incrementally* built, its lists are initially shorter and the join is faster. Further, the overall memory requirements are reduced since each T_{R_i} is constructed and processed separately, but most importantly, it can be discarded right after joining it with I_S.

As an additional contribution of our study, we reveal that ordering the set items in increasing order of their frequency (in contrast with decreasing frequency proposed in [24]) in fact improves query performance. Although such an ordering may lead to a larger prefix tree (compared to `PRETTI`), it dramatically reduces the number of candidates during query processing and enables our adaptive technique to achieve high performance gains.

We focus on main-memory evaluation of set containment joins (i.e., we optimize the main module of `PRETTI`, which joins two in-memory partitions); note that our solution is easily integrated in the block-based approaches of [24,27]. The fact that we limit the size of the prefix tree and that we use the `OPJ` paradigm, allows our method to operate with larger partitions compared to `PRETTI` in an external-memory problem, thus making our overall improvements even higher. Our thorough experimental evaluation using real datasets of different characteristics shows that our framework always outperforms `PRETTI`, being up to more than one order of magnitude times faster and saving at least 50% of memory.

Outline. The rest of the paper is organized as follows. Section 2 describes in detail the state-of-the-art set containment join algorithm `PRETTI`. Our adaptive evaluation methodology and the `OPJ` novel join paradigm are presented in Sections 3 and 4, respectively. Section 5 presents our experimental evaluation. Finally, Section 6 reviews related work and Section 7 concludes the paper.

2 Background on Set Containment Join: The `PRETTI` Algorithm

In this section, we describe in detail the state-of-the-art method `PRETTI` [24] for computing the $R \bowtie_{\subseteq} S$ set containment join of two collections R and S. The method has the following key features:

(i) The left-hand collection R is indexed by a prefix tree T_R and the right-hand collection S by an inverted index I_S. Both index structures are built on-the-fly, which enables the generality of the algorithm (for example, it can be applied for arbitrary data partitions instead of entire collections, and/or on data produced by underlying operators without interesting orders).

(ii) `PRETTI` traverses the prefix tree T_R in a depth-first manner. While following a path on the tree, the algorithm intersects the corresponding lists of inverted index I_S. The join algorithm is identical to the one proposed in [27] (see Section 6); however, due to grouping the objects under T_R, `PRETTI` performs the intersections for all sets in R with a common prefix only once.

r_1: $\{G, F, E, C, B\}$	s_1: $\{D, C, A\}$
r_2: $\{G, F, D, B\}$	s_2: $\{G, F, E, D, C, A\}$
r_3: $\{G, D, A\}$	s_3: $\{D, B\}$
r_4: $\{F, D, C, B\}$	s_4: $\{G, F, C, B\}$
r_5: $\{G, F, E\}$	s_5: $\{G, F, E, B\}$
r_6: $\{E, C\}$	s_6: $\{F, E, D, C, B\}$
r_7: $\{G, F, E\}$	s_7: $\{G, E, D, C, B\}$
	s_8: $\{G, E, D, C, B\}$
	s_9: $\{G, F, E, D\}$
	s_{10}: $\{G, F, E, D\}$
	s_{11}: $\{G, F\}$
	s_{12}: $\{G, F, E\}$

(a) left-hand collection R (b) right-hand collection S

Fig. 2 Example of two collections R and S

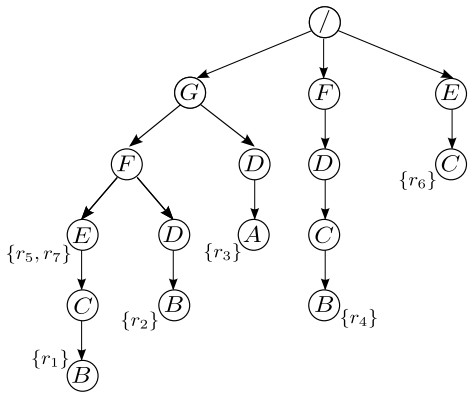
Algorithm 1: PRETTI(R, S)

input : Collections R and S; every object $r \in R$ is internally sorted such that the most frequent item appears first
output: the set J of all object pairs (r, s) such that $r \in R$, $s \in S$ and $r \subseteq s$

```
1  T_R ← ConstructPrefixTree(R);
2  I_S ← ConstructInvertedIndex(S);
3  foreach child node c of the root in T_R do
4  |  CL ← {s|s ∈ S};                                   // Candidates list
5  |  ProcessNode(c, CL, I_S, J);
6  return J;

7  Function ProcessNode(n, CL, I_S, J)
8  CL' ← CL ∩ I_S[n.item];                              // List intersection
9  foreach object r ∈ n.RL do
10 |  foreach object s ∈ CL' do
11 |  |  J ← J ∪ (r, s);
12 foreach child node c of n do
13 |  ProcessNode(c, CL', I_S, J);                      // Recursion
```

Algorithm 1 illustrates the pseudocode of PRETTI. During the initialization phase (Lines 1–2), PRETTI builds prefix tree T_R and inverted index I_S for input collections R and S, respectively. To construct T_R, every object r in R is internally sorted, so that its items appear in decreasing order of their frequency in R (this ordering is expected to achieve the highest path compression for T_R).[1] Each node n of prefix tree T_R is a triple $(item, path, RL)$ where $n.item$ is an item, $n.path$ is the sequence of the items in the nodes from the root of T_R to n (including $n.item$), and finally, $n.RL$ is the set of objects in R whose content is equal to $n.path$. For example, Figure 3(a) depicts prefix tree T_R for collection R in Figure 2(a). Set $n.RL$ is shown next to every node n unless it is empty. The inverted index I_S on collection S associates each item i in the domain of S to a *postings* list denoted

[1] Our experiments show that an increasing frequency order is in practice more beneficial. Yet, for the sake of readability, we present both PRETTI and our methodology considering a decreasing order.



(a) prefix tree T_R

A: $\{s_1,s_2\}$
B: $\{s_3, s_4,s_5,s_6,s_7,s_8\}$
C: $\{s_1,s_2,s_4,s_6,s_7,s_8\}$
D: $\{s_1,s_2,s_3,s_6,s_7,s_8,s_9,s_{10}\}$
E: $\{s_2,s_5,s_6,s_7,s_8,s_9,s_{10},s_{12}\}$
F: $\{s_2,s_4,s_5,s_6,s_9,s_{10},s_{11},s_{12}\}$
G: $\{s_2,s_4,s_5,s_7,s_8,s_9,s_{10},s_{11},s_{12}\}$

(b) inverted index I_S

Fig. 3 Indices of `PRETTI` for the collections in Figure 2

by $I_S[i]$. The $I_S[i]$ postings list has an entry for every object $s \in S$ that contains item i. Figure 3(b) pictures inverted index I_S for collection S in Figure 2(b).

The second phase of the algorithm involves the computation of the join result set J (Lines 3–5). `PRETTI` traverses the subtree rooted at every child node c of T_R's root by recursively calling the `ProcessNode` function. For a node n, `ProcessNode` receives as input from its parent node p in T_R, a candidates list CL. List CL includes all objects $s \in S$ that contain every item in $p.path$, i.e., $p.path \subseteq s$. Note that for every child of the root in T_R, $CL = S$. Next, `ProcessNode` intersects CL with inverted list $I_S[n.item]$ to find the objects in S that contain $n.path$ and stores them in CL' (Line 8). At this point, every pair of objects in $n.RL \times CL'$ is guaranteed to be a join result (Lines 9–11). Finally, the algorithm calls `ProcessNode` for every child node of n (Line 12–13).

Example 1 We demonstrate `PRETTI` *for the set containment join of collections R and S in Figure 2. The algorithm constructs prefix tree T_R and inverted index I_S shown in Figures 3(a) and 3(b), respectively. To construct T_R note that the items inside every object $r \in R$ are internally sorted in decreasing order of global item frequency in R (this is not necessary for the objects in S). First,* `PRETTI` *traverses the leftmost subtree of T_R under the node labeled by item G. Considering paths $\langle /, G\rangle$ and $\langle /, G, F\rangle$, the algorithm intersects candidates list CL (initially containing every object in S, i.e., $\{s_1, \ldots, s_{12}\}$) first with $I_S[G]$ and then with $I_S[F]$, and produces candidates list $\{s_2,s_4,s_5,s_9, s_{10},s_{11},s_{12}\}$, i.e., the objects in S that contain both G and F. The RL lists of the nodes examined so far are empty and thus, no result pair is reported. Next, path $\langle /, G, F, E\rangle$ is considered where CL is intersected with $I_S[E]$ producing $CL' = \{s_2,s_5,s_9,s_{10},s_{12}\}$. At current node, $RL = \{r_5,r_7\}$, and thus,* `PRETTI` *reports result pairs (r_5, s_2), (r_5, s_5), (r_5, s_9), (r_5, s_{10}), (r_5, s_{12}), (r_7, s_2), (r_7, s_5), (r_7, s_9), (r_7, s_{10}), (r_7, s_{12}). The algorithm proceeds in this manner to examine the rest of the prefix tree nodes performing in total* 15 *list intersections. The result of the join contains* 16 *pairs of objects.* ■

Finally, to deal with the case where the available main memory is not sufficient for computing the entire set containment join of the input collections, a partition-

based join strategy was also proposed in [24]. Particularly, the input collections R and S are horizontally partitioned so that the prefix tree and the inverted index for each pair of partitions (R_i, S_j) from R and S, respectively, fit in memory. Then, in a nested-loop fashion, each partition R_i is joined in memory with every partition S_j in S invoking `PRETTI`(R_i, S_j).

3 An Adaptive Methodology

By employing a prefix tree on the left-hand collection R, `PRETTI` avoids redundant intersections and thus outperforms previous methods that used only inverted indices, e.g., [27]. However, we observe two important shortcomings of the `PRETTI` algorithm. First, the cost of building and storing the prefix tree on R can be high especially if R contains sets of high cardinality. This raises a challenge when the available memory is limited which is only partially addressed by the partition-based join strategy in [24]. Second, after a candidates list CL becomes short, continuing the traversal of the prefix tree to obtain the join results for CL may incur many unnecessary in practice inverted list intersections. This section presents an adaptive methodology which builds upon and improves `PRETTI`. In Section 3.1 we primarily target the first shortcoming of `PRETTI` proposing the `LIMIT` algorithm, while in Section 3.2 we propose an extension to `LIMIT`, termed `LIMIT+`, that additionally deals with the second shortcoming.

3.1 The `LIMIT` Algorithm

To deal with the high building and storage cost of the prefix tree T_R, [24] suggests to partition R, as discussed in the previous section. Instead, we propose to build T_R only up to a predefined maximum depth ℓ, called *limit*. Hence, computing set containment join becomes a two-phase process that involves a *candidate generation* and a *verification* stage; for every candidate pair (r, s) with $|r| > \ell$ we need to compare the suffixes of objects r and s beyond ℓ in order to determine whether $r \subseteq s$. This approach is adopted by the `LIMIT` algorithm.

Algorithm 2 illustrates the pseudocode of `LIMIT`. Compared to `PRETTI` (Algorithm 1), `LIMIT` differs in two ways. First in Line 1, `LIMIT` constructs *limited* prefix tree ℓT_R on the left-hand collection R w.r.t. limit ℓ. The ℓT_R prefix tree has almost identical structure to *unlimited* T_R built by `PRETTI` except that the $n.RL$ list of a *leaf* node n contains every object $r \in R$ with $r \supseteq n.path$ instead of $r = n.path$. Figures 4(a) and (b) illustrate the limited versions of the prefix tree in Figure 3(b) for $\ell = 2$ and $\ell = 3$, respectively. Second, the `ProcessNode` function distinguishes between two cases of objects in $n.RL$ (Lines 11–14). If, for a object $r \in n.RL$, $|r| \leq \ell$ holds, then $r = n.path$ and, similar to `PRETTI`, pair (r, s) is guaranteed to be part of the join result J (Line 12). Otherwise, $r \supset n.path$ holds and `ProcessNode` invokes the `Verify` function which compares the suffixes of objects r and s beyond ℓ (Line 14). Intuitively, the latter case arises only for *leaf* nodes according to the definition of the *limited* prefix tree. To achieve a low verification cost, the objects of *both* R and S collections are internally sorted, i.e., the items appear in decreasing order of their frequency in $R \cup S$, which enables `Verify` to operate in a merge-sort manner.

Algorithm 2: LIMIT(R, S, ℓ)

input : Collections R and S, limit ℓ; every object $r \in R$ and $s \in S$ is internally sorted such that the most frequent item in $R \cup S$ appears first
output: the set J of all object pairs (r, s) such that $r \in R$, $s \in S$ and $r \subseteq s$

```
1  ℓT_R ← ContructPrefixTree(R, ℓ);
2  I_S ← ConstructInvertedIndex(S);
3  foreach child node c of the root in T_R do
4      CL ← {s|s ∈ S};                                   // Candidates list
5      ProcessNode(c, ℓ, CL, I_S, J);
6  return J;

7  Function ProcessNode(n, ℓ, CL, I_S, J)
8  CL' ← CL ∩ I_S[n.item];                               // List intersection
9  foreach object s ∈ CL' do
10     foreach object r ∈ n.RL do
11         if |r| ≤ ℓ then
12             J ← J ∪ (r, s);
13         else
14             Verify(r, s, ℓ, J);                       // Compare object suffixes
15 foreach child node c of n do
16     ProcessNode(c, ℓ, CL', I_S, J);                   // Recursion
```

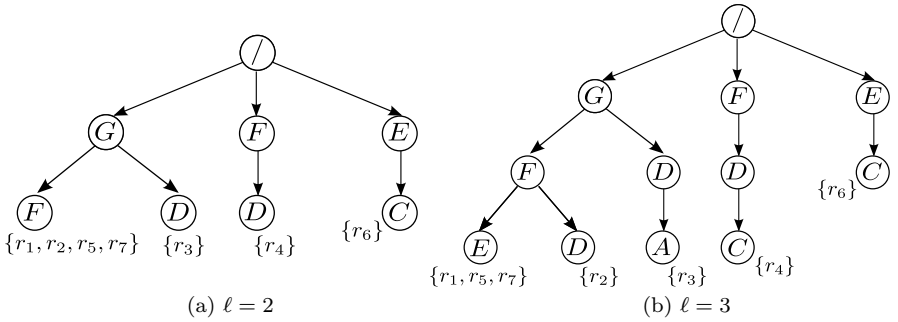


Fig. 4 *Limited* prefix tree ℓT_R for collection R in Figure 2

Example 2 We demonstrate LIMIT *using collections R and S in Figure 2; in contrast to* PRETTI *and Example 1, the objects of both collections are internally sorted. Consider first the case of $\ell = 2$.* LIMIT *constructs* limited *prefix tree ℓT_R shown in Figure 4(a) for collection R in Figure 2(a), and inverted index I_S in Figure 3(b). Then, similar to* PRETTI*, it traverses ℓT_R. When considering path $\langle /, G, F \rangle$, candidates list $CL' = \{s_2, s_4, s_5, s_9, s_{10}, s_{11}, s_{12}\}$ is produced. The $RL = \{r_1, r_2, r_5, r_7\}$ set of current node (F) is non-empty and thus, the algorithm examines every pair of objects from $RL \times CL'$ to report join results. As all objects in RL are of length larger than limit $\ell = 2$,* LIMIT *compares the suffixes beyond length $\ell = 2$ of all candidates by calling* Verify*, and finally, reports results (r_5, s_2), (r_5, s_5), (r_5, s_9), (r_5, s_{10}), (r_5, s_{12}), (r_7, s_2), (r_7, s_5), (r_7, s_9), (r_7, s_{10}), (r_7, s_{12}). At the next steps, the algorithm proceeds in a similar way to examine the rest of the prefix tree nodes performing 4 list intersections and verifying 37 candidate pairs by comparing their*

suffixes. Finally, if $\ell = 3$ `LIMIT` *traverses similarly prefix tree* ℓT_R *in Figure 4(b) performing* 8 *this time list intersections but verifying only* 10 *candidate object pairs by comparing their suffixes.* ■

The advantage of `LIMIT` over `PRETTI` and the partition-based join strategy of [24] is two-fold. First, building the prefix tree up to ℓ is faster than building the entire tree, but most importantly, with ℓ, the space needed to store the tree in main memory is reduced. If the *unlimited* T_R does not fit in memory, `PRETTI` would partition R and construct a separate (memory-based) T_{R_i} for each partition R_i; therefore, two objects r_i, r_j of R that have the same ℓ-prefix but belong to different partitions R_i and R_j, would be considered separately, which increases the evaluation cost of the join. In other words, reducing the size of T_R to fit in memory can have high impact on performance. In contrast, `LIMIT` guarantees that, for every path of length up to ℓ on *limited* ℓT_R, all redundant intersections are avoided similar to utilizing the *unlimited* prefix tree. Finally, an interesting aftermath of employing ℓ for set containment joins is related to the second shortcoming of `PRETTI`. For instance, with $\ell = 3$ and prefix tree ℓT_R in Figure 3(b), `LIMIT` will verify object r_1 against $CL = \{s_2, s_5, s_9, s_{10}, s_{12}\}$ and quickly determine that it is not part of the join result without performing two additional inverted list intersections.

An issue still open involves how limit ℓ is defined and most importantly, whether there is an optimal value of ℓ that balances the benefits of using the *limited* prefix tree over the cost of including a verification stage. Determining the optimal value for ℓ is a time-consuming task which involves more than an extra pass over the input collections. In specific, it requires computing expensive statistics with a process reminiscent to frequent itemsets mining; note that this process must take place online before building ℓT_R. Instead, in Section 5.4 we discuss and evaluate four strategies for estimating a good ℓ value based on simple and cheap-to-compute statistics. Our analysis shows that typically these strategies tend to overestimate the optimal ℓ. Besides, we also observe that the optimal ℓ value may in fact vary between different subtrees of ℓT_R depending on the number of objects stored inside the nodes. In view of this, we next propose an *adaptive* extension to `LIMIT` which employs an ad-hoc limit ℓ for each path of ℓT_R by dynamically choosing between list intersection and verification of the objects under the current subtree.

3.2 The `LIMIT+` Algorithm

As Example 2 shows, using limit ℓ for set containment joins introduces an interesting trade-off between list intersection and candidates verification which is directly related to the second shortcoming of the `PRETTI` algorithm. Specifically, as ℓ increases and `LIMIT` traverses longer paths of ℓT_R, candidates lists CL shorten due to the additional list intersections performed. Consequently, the number of object pairs to be verified by accessing their suffixes also reduces. However, from some point on, the number of candidates in CL no longer significantly reduces or, even worst, it remains unchanged; therefore, performing additional list intersections becomes a bottleneck. Similarly, if for a node n, CL is already too short, verifying the candidate pairs between the contents of CL and the objects contained under the subtree rooted at n can be faster than performing additional list intersections.

Algorithm 3: LIMIT+(R, S, ℓ)

input : Collections R and S, limit ℓ; every object $r \in R$ and $s \in S$ is internally sorted such that the most frequent item in $R \cup S$ appears first
output: the set J of all object pairs (r, s) such that $r \in R$, $s \in S$ and $r \subseteq s$

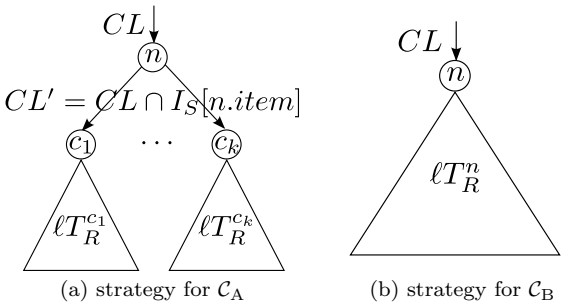
```
1  ℓT_R ← ContructPrefixTree(R, ℓ);
2  I_S ← ConstructInvertedIndex(S);
3  foreach child node c of the root in T_R do
4      CL ← {s|s ∈ S};                                   // Candidates list
5      ProcessNode(c, ℓ, CL, I_S, J);
6  return J;

7  Function ProcessNode(n, ℓ, CL, I_S, J)
8  if ContinueAsLIMIT(n, CL, I_S) then
9      CL' ← CL ∩ I_S[n.item];                           // List intersection
10     foreach object s ∈ CL' do
11         foreach object r ∈ n.RL do
12             if |r| ≤ ℓ then
13                 J ← J ∪ (r, s);
14             else
15                 Verify(r, s, ℓ, J);                   // Compare object suffixes
16     foreach child node c of n do
17         ProcessNode(c, ℓ, CL', I_S, J);               // Recursion
18 else
19     foreach object s ∈ CL do
20         foreach object r ∈ ℓT_R^n do                  // ℓT_R^n:subtree under n
21             Verify(r, s, ℓ−1, J);                     // Compare object suffixes
```

The LIMIT algorithm addresses only a few of the cases when candidates verification is preferred over list intersection, for instance the case of object r_1 in Figure 2(a) with limit $\ell = 3$. Due to *global* limit ℓ, the "blind" approach of LIMIT processes every path of the prefix tree in the same manner. To tackle this problem, we devise an *adaptive* strategy of processing ℓT_R adopted by the LIMIT+ algorithm. Apart from *global* limit ℓ, LIMIT+ also employs a dynamically determined *local* limit ℓ_p for each path p of the prefix tree. The basic idea behind this process is to decide on-the-fly for every node n of the prefix tree between:

(A) performing the $CL' = CL \cap I_S[n.item]$ intersection, reporting the pairs in $n.RL \times CL'$, and then, processing the descendant nodes of n in a similar way, or
(B) stopping the traversal of the current path and verifying the candidates between the objects of R contained in the subtree rooted at n denoted by ℓT_R^n and those in CL, i.e., all candidate pairs in $\ell T_R^n \times CL$.

In the first case, LIMIT+ would operate exactly as LIMIT does for the *internal* nodes of ℓT_R while in the second case, it would treat node n as a *leaf* node but without performing the corresponding list intersection. Therefore, in practice, a *local* limit for current path *n.path* is employed by LIMIT+.

Algorithm 3 illustrates the pseudocode of LIMIT+. Compared to LIMIT (Algorithm 2), LIMIT+ only differs on how a node of ℓT_R is processed. Specifically, given a node n, ProcessNode calls the ContinueAsLIMIT function (Line 8) to determine



(a) strategy for $\mathcal{C}_\mathrm{A}$ (b) strategy for $\mathcal{C}_\mathrm{B}$

Fig. 5 The two strategies considered by `LIMIT+`

whether the algorithm will continue processing n similar to `LIMIT` (Lines 10–17), or it will stop traversing current path $n.path$ and start verifying all candidates in $\ell T_R^n \times CL$ invoking the `Verify` function (Lines 18–21). In the latter case, notice that for every verifying pair (r, s) with $r \in \ell T_R^n \times CL$ and $s \in CL$, the algorithm accesses the suffixes of r and s beyond length $\ell - 1$ and not ℓ as the $CL \cap I_S[n.item]$ intersection has not taken place for current node n (Line 21).

Next, we elaborate on `ContinueAsLIMIT`. Intuitively, in order to determine how `LIMIT+` will process current node n the function has to first estimate and then compare the computational costs $\mathcal{C}_\mathrm{A}$ and $\mathcal{C}_\mathrm{B}$ of the two alternative strategies: (A) processing current node and its descendants in the subtree ℓT_R^n similar to `LIMIT`, or (B) verifying candidates in $\ell T_R^n \times CL$. In practice, it is not possible to estimate the cost of processing current node n and its descendants in ℓT_R^n similar to `LIMIT` since the involved intersections are not known in advance with the exception of $CL \cap I_S[n.item]$. Therefore, we estimate $\mathcal{C}_\mathrm{A}$ as the cost of computing the list intersection at current node n and, verifying, for each child node c_i of n, the candidate pairs between *all* objects under subtree $\ell T_R^{c_i}$ and the objects in CL'. Figure 5 illustrates the two alternative strategies, the costs of which are compared by `ContinueAsLIMIT`.

We now discuss how costs $\mathcal{C}_\mathrm{A}$ and $\mathcal{C}_\mathrm{B}$ can be estimated. For this purpose, we first break $n.RL$ set into two parts: $n.RL = n.RL^{=} \cup n.RL^{\supset}$, where $n.RL^{=}$ denotes the objects r in $n.RL$ with $r = n.path$, while $n.RL^{\supset}$ the objects with $r \supset n.path$. Note that according to the definition of *limited* prefix tree ℓT_R, $n.RL = n.RL^{=}$ holds for every *internal* node n, as $n.RL^{\supset} = \emptyset$. Second, we introduce the following cost functions to capture the computational cost of the three tasks involved in strategies (A) and (B):

(i) **List intersection**. The cost of computing $CL' = CL \cap I_S[n.item]$ in current node n, denoted by $\mathcal{C}_\cap$, depends on the lengths of the involved lists and it is also related to the way list intersection is actually implemented. For instance, if list intersection is performed in a merge-sort manner, then $\mathcal{C}_\cap$ is linear to the sum of the lists' length, i.e., $\mathcal{C}_\cap = \alpha_1 \cdot |CL| + \beta_1 \cdot |I_S[n.item]| + \gamma_1$. On the other hand, if the intersection is based on a binary search over the $I_S[n.item]$ list then $\mathcal{C}_\cap = \alpha_2 \cdot |CL| \cdot log_2(|I_S[n.item]|) + \beta_2$. Note that constants α_1, α_2, β_1, β_2 and γ_1 can be approximated by executing list intersection for several inputs and then, employing regression analysis over the collected measurements.

(ii) **Direct output of results.** Similar to `PRETTI` and `LIMIT`, after list intersection $CL' = CL \cap I_S[n.item]$, every pair (r, s) with $r \in n.RL$ and $s \in CL'$ such that $r = n.path$, i.e., $r \in n.RL^{=}$, is guaranteed to be among the join results and it would be directly reported. The cost of this task, denoted by $\mathcal{C}_\mathrm{d}$, is linear to the number of object pairs to be reported, and thus, $\mathcal{C}_\mathrm{d} = \alpha_3 \cdot |CL'| \cdot |n.RL^{=}| + \beta_3$. Constants α_3 and β_3 can be approximated by regression analysis.

(iii) **Verification.** To determine whether an (r, s) pair is part of the join result `Verify` would compare their suffixes in a merge-sort manner. Under this, the verification cost for each candidate pair is linear to the sum of their suffixes' length. Both alternative strategies considered by `ContinueAsLIMIT` involve verifying all candidate pairs between a subset of objects in R and a subset in S (candidates list CL or CL'). Without loss of generality consider the case of strategy (A). In total, $|\ell T_R^n \setminus n.RL^{=}| \cdot |CL'|$ candidates would be verified. Considering the length sum of the objects in ℓT_R^n and of the objects in CL', the total verification cost for (A) is

$$\mathcal{C}_\mathrm{v} = \alpha_4 \cdot |CL'| \cdot \sum_{r \in \{\ell T_R^n \setminus n.RL^{=}\}} (|r| - \ell) + \beta_4 \cdot |\ell T_R^n \setminus n.RL^{=}| \cdot \sum_{s \in CL'} (|s| - \ell) + \gamma_4$$

where $|r| - \ell$ ($|s| - \ell$) equals the length of the suffix for a object r (s) with respect to limit ℓ. Similar to the previous tasks, constants α_4, β_4 and γ_4 can be approximated by regression analysis. On the other hand, to approximate $|CL'| = |CL \cap I_S[n.item]|$ and $\sum_{s \in CL'} (|s| - \ell)$, we adopt an independent assumption approach based on the frequency of the item contained in current node n. Under this, $|CL'| \approx |CL| \cdot \frac{|I_S[n.item]|}{|S|}$ while the length sum of the objects in CL' can be estimated with respect to the $\frac{|CL'|}{|CL|} \approx \frac{|I_S[n.item]|}{|S|}$ decrease ratio, hence, we have $\sum_{s \in CL'} (|s| - \ell) \approx \frac{|I_S[n.item]|}{|S|} \cdot \sum_{s \in CL} (|s| - \ell)$. Finally, note that $\sum_{r \in \{\ell T_R^n \setminus n.RL^{=}\}} (|r| - \ell)$ can be computed using statistics gathered while building prefix tree ℓT_R and that $\sum_{s \in CL} (|s| - \ell)$ can be computed while performing the list intersection at the parent of current node n.

With $\mathcal{C}_\cap$, $\mathcal{C}_\mathrm{d}$, and $\mathcal{C}_\mathrm{v}$, the computational costs of the (A) and (B) strategies considered by `ContinueAsLIMIT` are estimated by:

$$\mathcal{C}_\mathrm{A} = \mathcal{C}_\cap(CL, I_S[n.item]) + \mathcal{C}_\mathrm{d}(n.RL^{=}, CL') + \mathcal{C}_\mathrm{v}(\{\ell T_R^n \setminus n.RL^{=}\}, CL', \ell)$$
$$\mathcal{C}_\mathrm{B} = \mathcal{C}_\mathrm{v}(\ell T_R^n, CL, \ell - 1)$$

As intersection $CL' = CL \cap I_S[n.item]$ is not computed in (B), candidates list CL and object suffixes beyond $\ell - 1$ are considered by $\mathcal{C}_\mathrm{B}$ in place of CL' and suffixes beyond ℓ considered by $\mathcal{C}_\mathrm{A}$.

Example 3 We illustrate the functionality of `LIMIT+` *using Example 2. Assuming* $\ell = 3$, `LIMIT+` *constructs prefix tree* ℓT_R *of Figure 4(b) and inverted index* I_S *of Figure 3(b). First, the algorithm traverses the subtree of* ℓT_R *under the node labeled by item G. The computational cost of the alternative strategies for this*

node are as follows. $\mathcal{C}_A$ *involves the cost of computing* $CL' = \{s_1, \ldots, s_{12}\} \cap I_S[G] = \{s_2, s_4, s_5, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$ *and based on the two child nodes, the cost of verifying all candidates in* $\{r_1, r_2, r_5, r_7\} \times CL'$ *and* $\{r_3\} \times CL'$*; note that no direct join results exist as RL for current node is empty. On the other hand,* $\mathcal{C}_B$ *captures the cost of verifying all candidates in* $\{r_1, r_2, r_3, r_5, r_7\} \times CL$*. Without loss of generality assume* $\mathcal{C}_A < \mathcal{C}_B$*. Hence,* LIMIT+ *processes current node (G) similar to* LIMIT*: path* $\langle /, G, F \rangle$ *and the node labeled by F are next considered. Assuming* $\mathcal{C}_A > \mathcal{C}_B$ *for this node,* LIMIT+ *imposes a local limit equal to* 2 *and verifies all candidates in* $\{r_1, r_2, r_5, r_7\} \times CL$ *with* $CL = \{s_2, s_4, s_5, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$ *(objects in S containing item G). Notice the resemblance to Example 2 for* $\ell = 2$ *with the exception that* $\{s_2, s_4, s_5, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\} \cap I_S[F]$ *is not computed.* ■

4 A Novel Join Paradigm

As discussed in Section 2, the join paradigm of PRETTI [24], which is also followed by LIMIT and LIMIT+, constructs the entire prefix tree T_R (or ℓT_R) and the entire inverted index I_S before joining them. However, we observe that the construction of T_R and I_S can be interleaved with the join process since for joining a set of objects from R that lie in a subtree of T_R it is not necessary to have constructed the entire I_S. For example, consider again the T_R and I_S indices of Figure 3. When performing the join for the nodes in the subtree rooted at node G, obviously, we need not have constructed the subtrees rooted at nodes F and E already. At the same time, only the objects from S that contain item G can be joined with each object in that subtree. Therefore, we only need a partially built I_S which includes just these objects. In this section, we propose a new paradigm, termed Order and Partition Join (OPJ), which is based on this observation. OPJ operates as follows:

(i) Assume that for each object (in either R or S), the items are considered in a certain order (i.e., in decreasing order of their frequency in $R \cup S$). OPJ partitions the objects of each collection into groups based on their first item.[2] Thus, for each item i, there is a partition R_i (S_i) of R (S) that includes all objects $r \in R$ ($s \in S$), for which the *first* item is i. For example, partition R_G of collection R in Figure 2(a) includes $\{r_1, r_2, r_3, r_5, r_7\}$, while partition R_E includes just r_6. Due to the internal sorting of the objects, an object in R_i or S_i includes i but does not include any item j, which comes before i in the order (e.g., $r_6 \in R_E$ cannot contain G or F). Then, OPJ initializes an empty inverted index I_S for S.

(ii) For each item i in order, OPJ creates a prefix tree T_{R_i} for partition R_i and updates I_S to include all objects from partition S_i. Then, T_{R_i} is joined with I_S using PRETTI (or our algorithms LIMIT and LIMIT+). After the join, T_{R_i} is dumped from the memory and OPJ proceeds with the next item $i + 1$ in order to construct $T_{R_{i+1}}$ using R_{i+1}, update I_S using S_{i+1} and join $T_{R_{i+1}}$ with I_S.

OPJ has several advantages over the PRETTI join paradigm. First, the entire T_R needs not be constructed and held in memory. For each item i the subtree of T_R

[2] This is different than the external-memory partitioning of the PRETTI paradigm, discussed at the end of Section 2.

Algorithm 4: `OPJ`(R, S, ℓ)

input : Collections R and S, limit ℓ; every Object $r \in R$ and $s \in S$ is internally sorted such that the most frequent item in $R \cup S$ appears first
output: the set J of all Object pairs (r, s) such that $r \in R$, $s \in S$ and $r \subseteq s$

```
1  Partition(S); Partition(R);            // w.r.t. the first item in each Object
2  I_S ← ∅;
3  foreach item i in decreasing frequency order do
4      ℓT_{R_i} ← ConstructPrefixTree(R_i, ℓ);
5      I_S ← UpdateInvertedIndex(I_S, S_i);
6      c ← child node of ℓT_{R_i}'s root;      // ℓT_{R_i}'s root has a single child c with
       c.item = i
7      CL ← Objects in S seen so far;                          // Candidates list
8      ProcessNode(c, CL, I_S, J, ℓ);                    // PRETTI,LIMIT,LIMIT+
9      delete ℓT_{R_i};
10 return J;
```

rooted at i (i.e., T_{R_i}) is built, joined, and then removed from memory. Second, the inverted index I_S is incrementally constructed, therefore T_{R_i} for each item i in order is joined with a smaller I_S which (correctly) excludes objects of S having only items that come after i. Thus, the inverted lists of the partially constructed I_S are shorter and the join is faster.[3] Finally, the overall memory requirements of `OPJ` are much lower compared to `PRETTI` join paradigm as `OPJ` only keeps one T_{R_i} in memory at a time (instead of the entire T_R).

Algorithm 4 illustrates a high-level sketch of the `OPJ` paradigm. `OPJ` receives as input collections R and S, and limit ℓ; for `PRETTI` $\ell = \infty$ (i.e., ℓT_{R_i} becomes T_{R_i}). Initially, collections R and S are partitioned to put all objects having i as their first item inside partitions R_i and S_i, respectively (Line 1). Also, I_S (the inverted index of S) is initialized (Line 2). Then, for each item i, `OPJ` computes the join results between objects from R having i as their first item and objects from S having i or a previous item in order as their first item (Lines 3–9). Specifically, for each item i in order, `OPJ` builds a (limited) prefix tree ℓT_{R_i} using partition R_i, adds all objects of partition S_i into I_S, and finally joins ℓT_{R_i} with I_S using the methodology of `PRETTI`, `LIMIT`, or `LIMIT+`. Note that for each ℓT_{R_i} the root has a single child c with $c.item = i$, because all objects in R_i have i as their first item. Thus, `OPJ` has to invoke the `ProcessNode` function (of either `PRETTI`, `LIMIT` or `LIMIT+`) only for c. In addition, note that candidates list CL is initialized with only the objects in S accessed so far instead of all objects in S according to the `PRETTI` join paradigm; the examination order guarantees that the rest of the objects in S cannot be joined with the objects in R under node c.

Example 4 We demonstrate `OPJ` *on collections R and S in Figure 2. The items in decreasing frequency order over $R \cup S$ are $G(14), F(13), E(12), D(11), C(9)$, $B(9)$, $A(3)$, resulting in the internally sorted objects shown in the figure. Without loss of generality, assume that the* `PRETTI` *algorithm is used to perform the join between each ℓT_{R_i} and I_S (i.e., $\ell = \infty$ and $\ell T_{R_i} = T_{R_i}$). Initially, the objects are partitioned according to their first item. The partitions for R are $R_G = \{r_1, r_2, r_3, r_5, r_7\}$, $R_F = \{r_4\}$, and $R_E = \{r_6\}$; the partitions for S are shown in Figure 6(a).* `OPJ` *first*

[3] Note that `OPJ` and `PRETTI` perform the same number of list intersections; i.e., `OPJ` does not save list intersections, but makes them cheaper.

S_G	s_2: $\{G, F, E, D, C, A\}$ s_4: $\{G, F, C, B\}$ s_5: $\{G, F, E, B\}$ s_7: $\{G, E, D, C, B\}$ s_8: $\{G, E, D, C, B\}$ s_9: $\{G, F, E, D\}$ s_{10}: $\{G, F, E, D\}$ s_{11}: $\{G, F\}$ s_{12}: $\{G, F, E\}$	A: $\{s_2\}$ B: $\{s_4,s_5,s_7,s_8\}$ C: $\{s_2,s_4,s_7,s_8\}$ D: $\{s_2,s_7,s_8,s_9,s_{10}\}$ E: $\{s_2,s_5,s_7,s_8,s_9,s_{10},s_{12}\}$ F: $\{s_2,s_4,s_5,s_9,s_{10},s_{11},s_{12}\}$ G: $\{s_2,s_4,s_5,s_7,s_8,s_9,s_{10},s_{11},s_{12}\}$
S_F	s_6: $\{F, E, D, C, B\}$	B: $\{s_4,s_5,s_6,s_7,s_8\}$ C: $\{s_2,s_4,s_6,s_7,s_8\}$ D: $\{s_2,s_6,s_7,s_8,s_9,s_{10}\}$ E: $\{s_2,s_5,s_6,s_7,s_8,s_9,s_{10},s_{12}\}$ F: $\{s_2,s_4,s_5,s_6,s_9,s_{10},s_{11},s_{12}\}$
S_D	s_1: $\{D, C, A\}$ s_3: $\{D, B\}$	A: $\{s_1,s_2\}$ B: $\{s_3,s_4,s_5,s_6,s_7,s_8\}$ C: $\{s_1,s_2,s_4,s_6,s_7,s_8\}$ D: $\{s_1,s_2,s_3,s_6,s_7,s_8,s_9,s_{10}\}$

(a) Partitions of S (b) Updates in I_S

Fig. 6 Employing the OPJ join paradigm

accesses partition R_G and builds T_{R_G}, which is identical to the leftmost subtree of the unlimited *T_R in Figure 3(a). Then,* OPJ *updates the (initially empty) inverted index I_S to include the objects of S_G; the resulting I_S is shown on the right of S_G, at the top of Figure 6(b). After joining T_{R_G} with I_S, T_{R_G} is deleted from memory, and the next item F in order is processed.* OPJ *builds T_{R_F} (which is identical to the 2nd subtree of T_R in Figure 3(a)) and updates I_S to include the objects in S_F; these updates are shown on the right of S_F in Figure 6(b). Then, T_{R_F} is joined with I_S, and* OPJ *proceeds to the next item E. In this case, T_{R_E} is built (the rightmost subtree of T_R in Figure 3(a)), but I_S is not updated as S_E is empty. Still, T_{R_E} is joined with current I_S. In the next round (item D), there is no join to be performed, because R_D is empty. If there were additional partitions R_i to be processed, I_S would have to be updated to include the objects in S_D, as shown on the right of S_D in Figure 6(b). However, since all objects from R have been processed,* OPJ *can terminate without processing S_D.* ■

5 Experimental Evaluation

In this section, we present an experimental evaluation of our methodology for set containment joins. Section 5.1 details the setup of our analysis. Section 5.2 investigates the preferred global ordering of the items, while Section 5.3 demonstrates the advantage of the OPJ join paradigm. Section 5.4 shows how limit ℓ affects the efficiency of our methodology and presents four strategies for estimating its optimal value. Finally, Section 5.5 conducts a performance analysis of our methods against the state-of-the-art PRETTI [24].

Table 1 Characteristics of real datasets

characteristic	BMS	FLICKR	KOSARAK	NETFLIX
Cardinality	$515K$	$1.7M$	$990K$	$480K$
Domain size	$1.6K$	$810K$	$41K$	$18K$
Avg object length	63	52	398	1,557
Weighted avg object length	7	10	9	210
Max object length	164	102	2497	17,653
File size (Mb)	11	76	31	407

Table 2 Characteristics of synthetic datasets

characteristic	**values**	**default value**	**file size (Gb)**
Cardinality	$1M$, $3M$, $5M$, $7M$, $10M$	$5M$	0.3, 0.8, 1.4, 1.9, 2.7
Domain size	$10K$, $50K$, $100K$, $500K$, $1M$	$100K$	1.1, 1.3, 1.4, 1.6, 1.6
Weighted avg object length	10, 30, 50, 70, 100	50	0.3, 0.8, 1.4, 1.9, 2.7
Zipfian distribution	0, 0.3, 0.5, 0.7, 1	0.5	1.4, 1.4, 1.4, 1.3, 1.1

5.1 Setup

Our experimental analysis involves both real and synthetic collections. Particularly, we use the following real datasets:

- BMS is a collection of click-stream data from Blue Martini Software and KDD 2000 cup [43].
- FLICKR is a collection of photographs from Flickr website for the city of London [10]. Each object contains the union of "tags" and "title" elements.
- KOSARAK is a collection of click-stream data from a hungarian on-line news portal available at http://fimi.ua.ac.be/data/.
- NETFLIX is a collection of user ratings on movie titles over a period of 7 years from the Netflix Prize and KDD 2007 cup.

Table 1 summarizes the characteristics of the real datasets. BMS covers the case of small domain collections while FLICKR the case of datasets with very large domains. NETFLIX is a collection of extremely long objects. In addition, to study the scalability of the methods, we generated synthetic datasets with respect to (i) the collection cardinality, (ii) the domain size, (iii) the weighted average object length and (iv) the order of the Zipfian distribution for the item frequency. Table 2 summarizes the characteristics of the synthetic collections. On each test, we vary one of the above parameters while the rest are set to their default values.

Similar to [24] for set containment joins (and other works on set similarity joins [9,41]), our experiments involve only self-joins, i.e., $R = S$ (note, however, that our methods operate exactly as in case of non self-joins, i.e., they take as input two copies of the same dataset). The collections and the indexing structures used by all join methods are stored entirely in main memory; as discussed in the introduction we focus on the main module of the evaluation methods which joins two in-memory partitions, but our proposed methodology is easily integrated in the block-based

approaches of [24, 27]. Further, we do not consider any compression techniques, as they are orthogonal to our methodology.

To assess the performance of each method, we measure its response time, the total number of intersections performed and the total number of candidates; note that the response time includes both the indexing and joining cost of the method, and in case of the `OPJ` paradigm, also the cost of sorting and partitioning the inputs. Finally, all tested methods are written in C++ and the evaluation is carried out on an 3.6Ghz Intel Core i7 CPU with 64GB RAM running Debian Linux.

5.2 Items Global Ordering

The goal of the first experiment is to determine the most appropriate ordering for the items inside an object. In practice, only the characteristics of prefix tree T_R and how it is utilized are affected by how we order the items inside each object (neither the size of inverted index I_S nor the number of objects accessed from S depend on this ordering). Therefore, in this experiment, we only focus on the `PRETTI` join paradigm. In [24], to construct a compact prefix tree T_R the items inside an object are arranged in decreasing order of their frequency. On the other hand, arranging the items in increasing frequency order allows for faster candidate pruning as the candidates list CL rapidly shrinks after a small number of list intersections. In other words, the ordering of the items affects not only the building cost and the storage requirements of T_R, but most importantly, the response time of the join method. In practice, we observe that the best ordering is also related to how the $CL \cap I_S[n.item]$ list intersection is implemented. Although the problem of list intersection is out of scope of this paper per se, we implemented: (i) a merge-sort based approach, and (ii) a hybrid approach based on [4] that either adopts the merge-sort approach or binary searches every object of CL inside the $I_S[n.item]$ postings list. Table 3 confirms our claim regarding the correlation between the global ordering of the items and the response time of the `PRETTI` join algorithm (note that the reported time involves both the indexing and the join phase of the method). Arranging the items in decreasing order of their frequency is generally better only if the merge-sort based approach is adopted for the list intersections, while in case of the hybrid approach, the objects should be arranged in increasing order; an exception arises for NETFLIX where adopting the increasing ordering is always more beneficial because of its extremely long objects. In summary, the combination of the hybrid approach and the increasing frequency global ordering minimizes the response time of the `PRETTI` algorithm in all cases. Thus, for the rest of this analysis, we employ the hybrid approach for list intersection and arrange the items inside an object in the increasing order of their frequency. Note that for matters of reference and completion we also include the original version of [24] denoted by `orgPRETTI` corresponding to the Decreasing-Hybrid combination of Table 3.

5.3 Employing the `OPJ` Join Paradigm

Next, we investigate the advantage of `OPJ` (Section 4) over the `PRETTI` join paradigm of [24]. For this purpose we devise an extension to the `PRETTI` algorithm that fol-

Table 3 Determining items global ordering, response time (sec) of the PRETTI algorithm

Dataset	Increasing		Decreasing	
	Merge-sort	Hybrid	Merge-sort	Hybrid
BMS	407	42	106	71
FLICKR	1606	30	187	108
KOSARAK	1606	73	282	136
NETFLIX	18,399	504	35,169	14,051

Table 4 Employing the OPJ join paradigm, response time (sec)

Dataset	orgPRETTI	PRETTI	PRETTI∗	Improvement ratio over	
				orgPRETTI	PRETTI
BMS	71	42	28	2.5×	1.5×
FLICKR	108	30	20	5.4×	1.5×
KOSARAK	136	73	54	2.5×	1.4×
NETFLIX	14,051	504	391	38.5×	1.3×

Table 5 Limit ℓ determined by each estimation strategy

Dataset	Optimal	*AVG*	*W–AVG*	*MDN*	*FRQ*
BMS	2	63	7	4	4
FLICKR	2	52	10	8	3
KOSARAK	4	398	9	3	5
NETFLIX	6	1,557	210	96	6

lows OPJ, denoted by PRETTI∗. Table 4 reports the response time of the algorithms. The results experimentally prove the superiority of the OPJ paradigm; PRETTI∗ is from 1.3 to 1.5 times faster than PRETTI. Recall at this point that compared to the algorithm discussed in [24], our version of PRETTI arranges the items in increasing order of their frequency as discussed in Section 5.2; thus, the overall improvement of PRETTI∗ (which follows OPJ) over the original method of [24] orgPRETTI is even greater: 2.5× for BMS-POS, 5.4× for FLICKR, 2.5× for KOSARAK and 38.5× for NETFLIX. For the rest of our analysis we adopt the OPJ paradigm for all tested methods.

5.4 The Effect of Limit ℓ

As discussed in Section 3, employing limit ℓ for set containment joins introduces a trade-off between list intersection and candidates verification. To demonstrate this effect, we run the LIMIT algorithm (adopting OPJ) while varying limit ℓ from 1 to the average object length in R, and then plot its response time (Figure 7), the number of list intersections performed (Figure 8) and the total number of candidates (Figure 9). The total number of candidates includes both (r, s) pairs which are directly reported as results, i.e., with $|r| \leq \ell$, and those that are verified by comparing their prefixes beyond ℓ, i.e., with $|r| > \ell$. To have a better understanding of this experiment we also include the measurements for PRETTI∗ which uses an *unlimited* T_R. The figures clearly show the trade-off introduced by limit ℓ and confirm the existence of an optimal value that balances the benefits of using

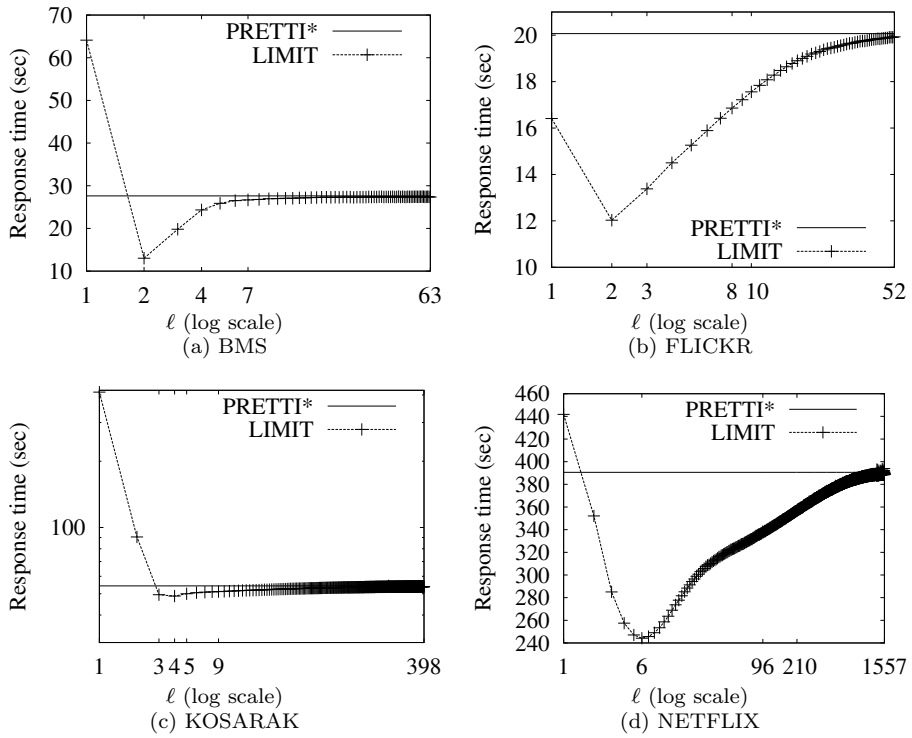


Fig. 7 Vary limit ℓ, response time

the *limited* prefix tree over the cost of including a verification stage. According to Figures 8 and 9, as ℓ increases, `LIMIT` naturally performs more list intersections, and thus, the number of candidate pairs decreases until it becomes equal to the join results, i.e., the number of candidates for `PRETTI`$*$. However, regarding its performance shown in Figure 7, although `LIMIT` initially benefits from having to verify fewer candidate pairs, when ℓ increases beyond a specific value, performing additional list intersections becomes a bottleneck and the algorithm slows down until its response time becomes almost equal to the time of `PRETTI`$*$.

Apart from the trade-off introduced by limit ℓ, Figures 7, 8 and 9 also show that the `LIMIT` algorithm can be faster than `PRETTI`$*$ as long as ℓ is properly set, i.e., close to its optimal value. However, as discussed in Section 3, determining the optimal ℓ value is a time-consuming procedure, reminiscent to frequent itemsets mining which cannot be employed in practice; recall that ℓ must be determined online. For this purpose, we propose the following simple strategies to select a good ℓ value based on cheap-to-compute statistics that require no more than a pass over the input collection R. First, strategies AVG and $W\text{–}AVG$ set ℓ equal to the average and the weighted average object length in R, respectively. Similarly, strategy MDN sets ℓ to the median value of the object length in R. Last, we also devise a frequency-based strategy termed FRQ. The idea behind FRQ is to estimate when paths greater than ℓ would only be contained in very few objects. We start with a path p that contains the most frequent item in R and progressively

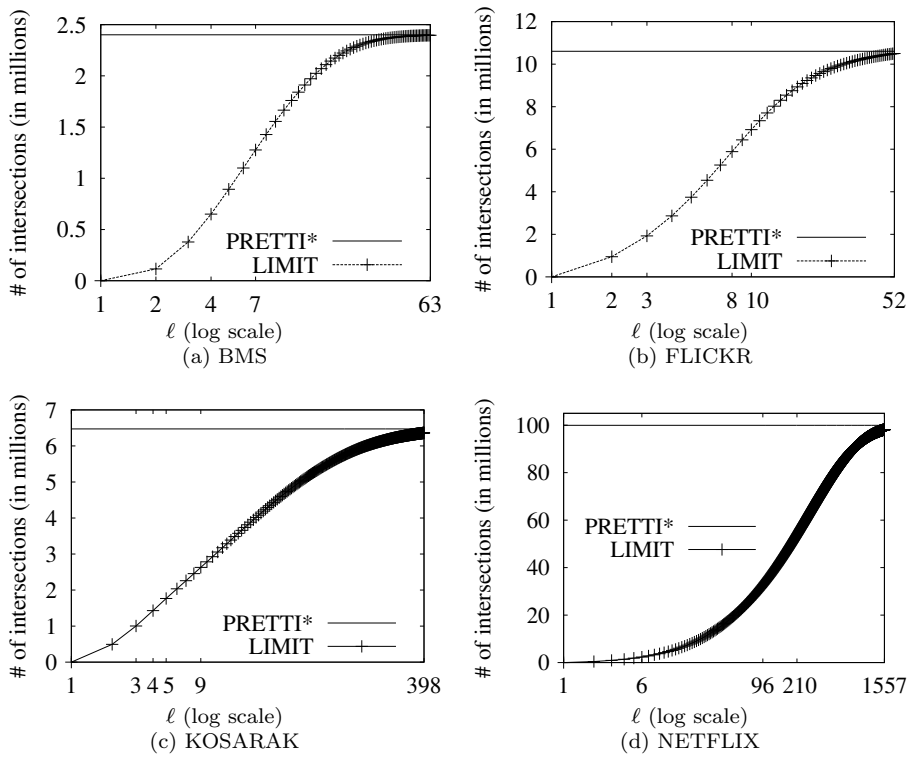


Fig. 8 Vary limit ℓ, number of intersections.

add the next items in decreasing frequency order. We estimate the probability that this path appears in a object by considering only the support of the items. When this probability falls under a threshold, which makes the expected cost of list intersection greater than the cost of verification (according to our analysis in Section 3.2), we stop adding items in p and set $\ell = |p|$. Note that this probability serves as an upper bound for all paths of length ℓ (assuming item independence), since p includes the most frequent items. Table 5 summarizes the values of ℓ determined by each strategy for the experimental datasets. Overall FRQ provides the best estimation of optimal ℓ; in fact for NETFLIX it identifies the actual optimal value. Figures 7, 8 and 9 confirm this observation as the performance of `LIMIT` with a limit set by FRQ is very close to its performance for the optimal ℓ. Thus, for the rest of our analysis we adopt FRQ to set limit ℓ value.

5.5 Comparison of the Join Methods

In Section 5.4, we showed that by properly selecting limit ℓ (FRQ strategy), `LIMIT` outperforms `PRETTI`$*$ and, based on Sections 5.3 and 5.2, also `PRETTI` and `orgPRETTI`. Next, we experiment with `LIMIT`+ which (like `LIMIT`) employs FRQ. Figure 10 reports the response time of `orgPRETTI`, `PRETTI`, `PRETTI`$*$, `LIMIT` and `LIMIT`+ on all four real datasets. To further investigate the properties of `LIMIT`+,

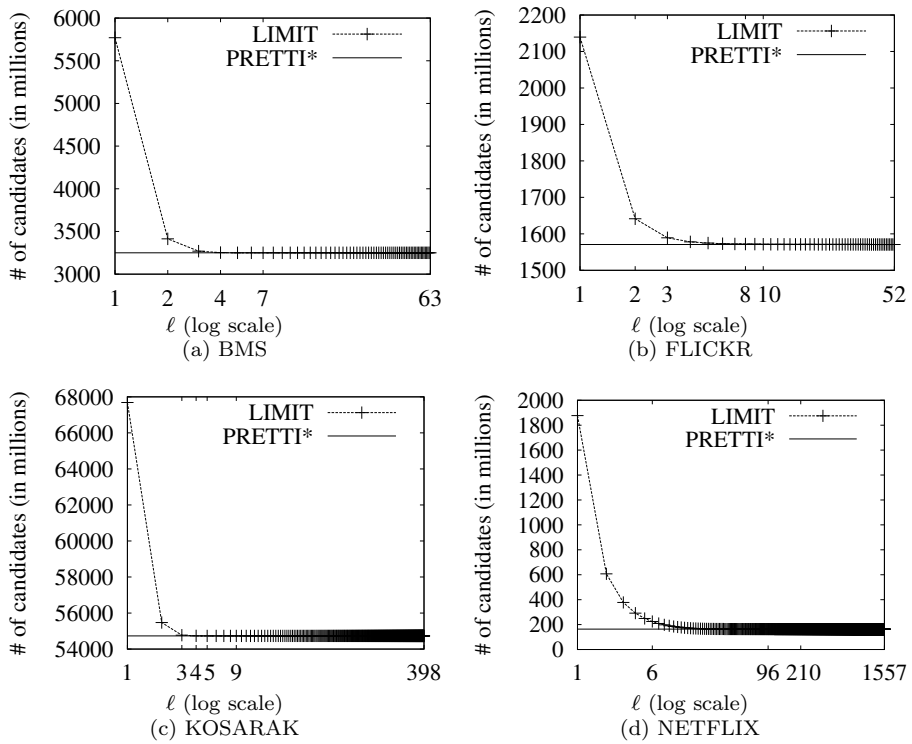


Fig. 9 Vary limit ℓ, number of candidates (for PRETTI$*$ equals the number of results)

we also include the response time of two oracle methods[4]: (i) L−ORACLE corresponds to LIMIT with ℓ set to its optimal value (see Table 5), (ii) T−ORACLE is a version of LIMIT+ which compares the actual execution time of the two alternative strategies for current prefix tree node instead of utilizing the cost model of Section 3.2; note that for this purpose we run offline both alternative strategies for every prefix tree node and store their execution time. With the exception of orgPRETTI and PRETTI the rest of the algorithms follow the OPJ join paradigm. We break the response time of all methods into three parts, (i) building prefix tree T_R, (ii) building inverted index I_S and (iii) computing the join results. Note that for PRETTI+, LIMIT, LIMIT+ and the oracles, the indexing time additionally includes the sorting and partitioning cost of the input objects. As expected the total indexing time is negligible compared to the joining time; an exception arises for FLICKR due its large number of objects.

Figure 10 shows that LIMIT+ is the most efficient method for set containment joins. It is at least two times faster than PRETTI. LIMIT+ also outperforms LIMIT for the BMS, FLICKR and KOSARAK datasets while for NETFLIX, both algorithms perform similarly as (i) the FRQ strategy sets limit ℓ to its optimal value and (ii) the T_R prefix tree for NETFLIX is quite balanced. The adaptive approach

[4] These are *infeasible* methods using apriori knowledge which is not known at runtime and it is extremely expensive to compute before the join.

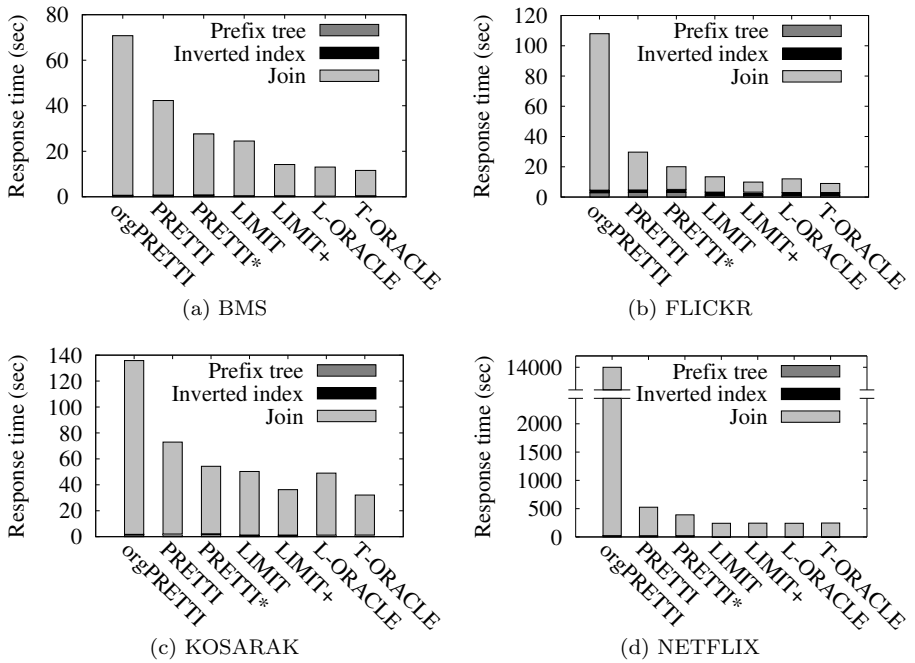


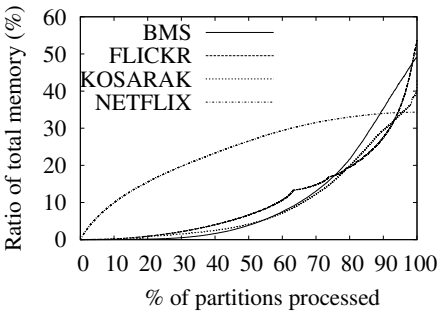
Fig. 10 Comparison of the set containment join methods on real datasets (limit ℓ set by FRQ according to Table 5)

of LIMIT+ that dynamically chooses between list intersection and candidates verification, copes better with (i) overestimated ℓ values and (ii) cases where T_R is unbalanced. Specifically, due to employing an ad-hoc limit for each path of the prefix tree, LIMIT+ can be faster than LIMIT even with optimal ℓ, i.e., faster than L−ORACLE (see Figures 10(b) and (c)). For these datasets, T_R is quite unbalanced and thus, there is no fixed value of ℓ to outperform the adaptive strategy. Note that even if ℓ is overestimated, e.g., using strategy $W\text{–}AVG$, the performance of LIMIT+ is almost the same as when an optimal (or close to optimal) ℓ is used. Note also that the response time of LIMIT+ is very close to that of T−ORACLE which proves the accuracy of our cost model proposed in Section 3.2. We would like to stress at this point that the overall performance improvement achieved by LIMIT+ over the original method of [24] which arranges the items inside an object in decreasing frequency order is as expected even larger compared to our version of PRETTI; LIMIT+ is 5 times faster than orgPRETTI for BMS, 11 times for FLICKR, 3.5 times for KOSARAK and 70 times for NETFLIX.

Next, we analyze the advantage of LIMIT+ (using FRQ) over orgPRETTI of [24] that arranges the items in decreasing frequency order, with respect to their memory requirements. Figure 11(a) shows the space for indexing only the left-hand collection R when neither method follows the OPJ paradigm. We observe that by constructing *limited* prefix tree ℓT_R instead of *unlimited* T_R, LIMIT+ saves at least 50% of space compared to orgPRETTI; for NETFLIX, where T_R has the highest storing cost due to its extremely long objects, the savings are over 90%. Then, in

Dataset	memory ratio $\ell T_R/T_R$
BMS	50%
FLICKR	44%
KOSARAK	46%
NETFLIX	3%

(a) LIMIT+ (not OPJ) Vs orgPRETTI

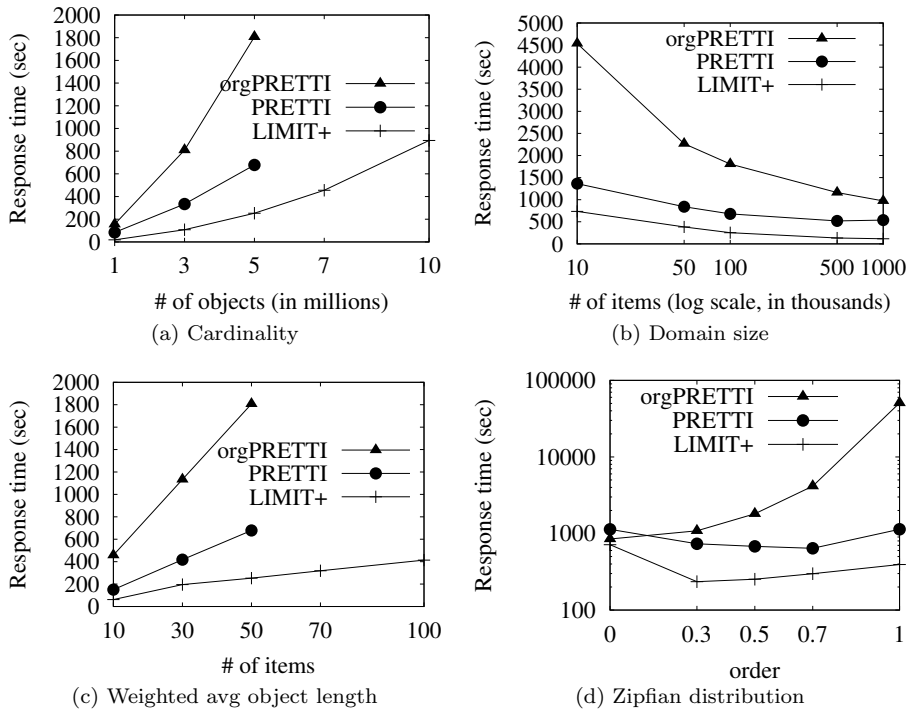


(b) LIMIT+ (OPJ) Vs orgPRETTI

Fig. 11 Memory requirements (LIMIT+ using FRQ)

Figure 11(b) we consider LIMIT+ adopting OPJ and report the space for indexing both input collections while evaluating the join, compared to orgPRETTI which does not follow the OPJ paradigm. We observe that by incrementally building ℓT_R and I_S, LIMIT+ uses at least 50% less space than orgPRETTI. Naturally, the amount of space used by LIMIT+ increases while examining the collection partitions, but it is always lower than the space for orgPRETTI due to never actually building and storing the entire prefix tree; only one subtree of ℓT_R is kept in memory at a time. Finally, notice the different trend for NETFLIX as its partitions have balanced sizes; in contrast for BMS, FLICKR and KOSARAK, the first partitions contain very few objects while the last ones are very large.

Finally, we present the results of our scalability tests on the synthetic datasets of Table 2. Figure 12 reports the response time of our best method LIMIT+ and the orgPRETTI and PRETTI competitors. The purpose of these tests is twofold: (i) to demonstrate how the characteristics of a dataset affect the performance of the methods, and (ii) to determine their "breaking point". First, we notice that all methods are affected in a similar manner; their response time increases as the input contains more or longer objects and decreases while the domain size becomes larger. An exception arises in Figure 12(d). The performance of orgPRETTI is severely affected when increasing the order of the Zipfian distribution; recall that orgPRETTI arranges the items inside an object, in decreasing frequency order. As expected, LIMIT+ outperforms orgPRETTI and PRETTI under all setups, similar to the case of real datasets. Second, we also observe that both orgPRETTI and PRETTI are unable to cope with the increase of the cardinality and weighted average object length of the datasets. These two factors directly affect the size of the T_R prefix tree and the memory requirements. In practice, orgPRETTI and PRETTI failed to run for inputs with more than $5M$ objects and/or when their weighted average length is larger than 50, because the *unlimited* prefix tree cannot fit inside the available memory; in these cases the methods would have to adopt a block-based evaluation approach similar [24, 27]. In contrast, LIMIT+ is able to index left-hand relation R due to employing limit ℓ and following OPJ, and hence, compute the join results.



(a) Cardinality (b) Domain size

(c) Weighted avg object length (d) Zipfian distribution

Fig. 12 Scalability tests on synthetic datasets (limit ℓ set by FRQ), default parameter values: candinality $5M$ objects, domain size $100K$ items, weighted avg object length 50 items, order of Zipfian distribution 0.5

6 Related Work

Our work is related to query operators on sets. In this section, we summarize previous work done for set containment queries, set containment joins, and set similarity joins. In addition, we review previous work on efficient computation of list intersection, which is a core module of our algorithms.

6.1 Set Containment Queries

Signatures and inverted files are two alternative indexing structures for set-valued data. Signatures are bitmaps used to exactly or approximately represent sets. With $|D|$ being the cardinality of the items domain, a set x is represented by a $|D|$-length signature $sig(x)$. The i-th bit of $sig(x)$ is set to 1 iff the i-th item of domain D is present in x. If the sets are very small compared to $|D|$, exact signatures are expensive to store, and therefore, approximations of fixed length $l < |D|$ are typically used. Experimental studies [22, 44] showed that inverted files outperform signature-based indices for set containment queries on datasets with low cardinality set objects, e.g., typical text databases.

In [37, 38], the authors proposed extensions of the classic inverted file data structure, which optimize the indexing set-valued data with skewed item distribu-

tions. In [14], the authors proposed an indexing scheme for text documents, which includes inverted lists for frequent word combinations. A main-memory method for addressing error-tolerant set containment queries was proposed in [1]. In [42], Zhang et al. addressed the problem of probabilistic set containment, where the contents of the sets are uncertain. The proposed solution relies on an inverted file where postings are populated with the item's probability of belonging to a certain object. The study in [23] focused on containment queries on nested sets, and proposes an evaluation mechanism that relies on an inverted file which is populated with information for the placement of an element in the tree of nested sets. The above methods use classic inverted files or extend them either by trading update and creation costs for response time [1,14,37,38] or by adding information that is needed for more complex queries [23,42]. Employing these extended inverted files for set containment joins (i.e., in place of our I_S) is orthogonal to our work.

6.2 Set Containment Joins

In [21], the *Signature Nested Loops* (`SNL`) Join and the *Signature Hash* Join (`SHJ`) algorithm for set containment joins were proposed, with `SHJ` shown to be the fastest. For each set object r in the left-hand collection R, both algorithms compare signatures to identify every object s in the right-hand collection S with $sig(r)$ & $\neg sig(s) = 0$ and $|r| \leq |s|$ (filter phase), and then, perform explicit set comparison to discard false drops (verification phase). Later, the hash-based algorithms *Partitioned Set Join* (`PSJ`) in [30] and *Divide-and-Conquer Set Join* (`DCJ`) in [28] aimed at reducing the quadratic cost of the algorithms in [21]. In these approaches, the input collections are partitioned based on hash functions such that object pairs of the join result fall in the same partition. Finally, Melnik and Molina [29] proposed adaptive extensions to `PSJ` and `DCJ`, termed `APSJ` and `ADCJ`, respectively, to overcome the problem of a potentially poor partitioning quality.

Inverted files were employed by [24,27] for set containment joins. Specifically, in [27], Mamoulis proposed a *Block Nested Loops* (`BNL`) Join algorithm that indexes the right-hand collection S by an inverted file I_S. The algorithm iterates through each object r in the left-hand collection R and intersects the corresponding postings lists of I_S to identify the objects in S that contain r. The experimental analysis in [27] showed that `BNL` is significantly faster than previous signature-based methods [21,30]. In [24], Jampani and Pudi targeted the major weakness of `BNL`; the fact that the overlaps between set objects are not taken into account. The proposed algorithm `PRETTI`, employs a prefix tree on the left-hand collection, allowing list intersections for multiple objects with a common prefix to be performed just once. Experiments in [24] showed that `PRETTI` outperforms `BNL` and previous signature-based methods of [29,30]. Our work first identifies and tackles the shortcomings of the `PRETTI` algorithm and then, proposes a new join paradigm.

6.3 Set Similarity Joins

The set similarity join finds object pairs (r, s) from input collections R and S, such that $sim(r, s) \geq \theta$, where $sim(\cdot, \cdot)$ is a similarity function (e.g., Jaccard

coefficient) and θ is a given threshold. Computing set similarity joins based on inverted files was first proposed in [34]: for each object in one input, e.g., $r \in R$, the inverted lists that correspond to r's elements on the other collection are scanned to accumulate the overlap between r and all objects $s \in S$. Among the optimization techniques on top of this baseline, Chaudhuri et al. [15] proposed a filter-refinement framework based on *prefix filtering*; for two internally sorted set objects r and s to satisfy $sim(r, s) \geq \theta$ their prefixes should have at least some minimum overlap. Later, [3,9,33,41] built upon prefix filtering to reduce the number of candidates generated. Recently, Bouros et al. [10] proposed a grouping optimization technique to boost the performance of the method in [41], and Wang et al. [40] devised a cost model to judiciously select the appropriate prefix for a set object. An experimental comparison of set similarity join methods can be found in [25]. In theory, the above methods can be employed for set containment joins, considering for instance the asymmetric containment Jaccard measure, $sim(r, s) = \frac{|r \cap s|}{|r|}$ and threshold $\theta = 1$. In practice, however, this approach is not efficient as it generates a large number of candidates. For each object $r \in R$ prefix filtering can only prune objects in S that do not contain r's first item while the rest of the candidates need to be verified by comparing the actual set objects. Therefore, the ideas proposed in previous work on set similarity joins are not applicable to set containment joins.

6.4 List Intersection

In [19,20], Demaine et al. presented an adaptive algorithm for computing set intersections, unions and differences. Specifically, the algorithm in [19] (ameliorated in [20] and extended in [7]) polls each list in a round robin fashion. Baeza-Yates [4] proposed an algorithm that adapts to the input values and performs quite well in average. It can be seen as a natural hybrid of the binary search and the merge-sort approach. Experimental comparison of the above, among others, methods of list intersection, with respect to their CPU cost can be found in [5,6,8]. The trade-off between the way sets are stored and the way they are accessed in the context of the intersection operator was studied in [18]. Finally, recent work [35,36,39] considered list intersection with respect to the characteristics of modern hardware and focused on balancing the load between multiple cores. In [35,36], Tatikonda et al. proposed inter-query parallelism and intra-query parallelism. The former exploits parallelism between different queries, while the latter parallelizes the processing within a single query. On the other hand, the algorithm in [39] probes the lists in order to gather statistics that would allow efficient exploration of the multi-level cache hierarchy. Efficient list intersection is orthogonal to our set containment join problem. Yet, in Section 5.2, we employ a hybrid list intersection method based on [4] to determine the preferred ordering of the items inside the objects.

6.5 Estimating Set Intersection Size

Estimating the intersection size of two sets has received a lot of attention in the area of information retrieval [11,12,16,17,26], to determine the similarity between two documents modelled as sets of terms. Given sets A and B, the basic idea is to compute via sampling small *sketches* $\mathcal{S}(A)$ and $\mathcal{S}(B)$, respectively. Then,

$|\mathcal{S}(A) \cap \mathcal{S}(B)|$ is used as an estimation of $|A \cap B|$. Our adaptive methodology for set containment joins (Section 3.2) involves estimating the size of a list intersection. Yet, the methods discussed above are not applicable as they require an expensive preprocessing step, i.e., precomputing and indexing the sketches for every list of the inverted index at the right-hand collection. In addition, one of the two lists at each intersection (i.e., candidates list CL) is the result of previous intersections. Thus, computing the sketch of CL should be done on-the-fly, i.e., the overall cost of the sketch-based intersection would exceed the cost of performing the exact list intersection (especially since CL becomes shorter every time it is intersected with a inverted list of the right-hand collection).

7 Conclusion

In this paper we revisited the set containment join $R \bowtie_{\subseteq} S$ between two collections R and S of set objects r and s, respectively. We presented a framework which improves the state-the-art method `PRETTI`, greatly reducing the space requirements and time cost of the join. Particularly, we first proposed an adaptive methodology (algorithms `LIMIT` and `LIMIT`+) that limits the prefix tree constructed for the left-hand collection R. Second, we proposed a novel join paradigm termed `OPJ` that partitions the objects of each collection based on their first contained item, and then examines these partitions to evaluate the join while progressively building the indices on R and S. Finally, we conducted extensive experiments on real datasets to demonstrate the advantage of our methodology.

Besides the fact that the `OPJ` paradigm significantly reduces both the join cost and the maximum memory requirements, it can be applied in a parallel processing environment. For instance, by assigning each partition R_i of the left-hand collection to a single computer node v_i while replicating the partitions of the right-hand collection such that node v_i gets every object in S which starts either by item i or an item before i according to the global item ordering, our method runs at each node and there is no need for communication among the nodes, since join results are independent and there are no duplicates. In the future, we plan to investigate the potential of such an implementation.

References

1. P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD Conference*, pages 927–938, 2010.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
3. A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
4. R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *CPM*, pages 400–408, 2004.
5. R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pages 13–24, 2005.
6. R. A. Baeza-Yates and A. Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications, Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, pages 45–61. 2010.
7. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, pages 390–399, 2002.

8. J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14:7:3.7–7:3.24, Jan. 2009.
9. R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
10. P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.
11. A. Broder. On the resemblance and containment of documents. In *SEQUENCES*, pages 21–29, 1997.
12. A. Z. Broder. Identifying and filtering near-duplicate documents. In *CPM*, pages 1–10, 2000.
13. B. Cao and A. Badia. A nested relational approach to processing sql subqueries. In *SIGMOD Conference*, pages 191–202, 2005.
14. S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, pages 663–670, 2007.
15. S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
16. Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *PODS*, pages 216–225, 2000.
17. Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Generalized substring selectivity estimation. *J. Comput. Syst. Sci.*, 66(1):98–132, 2003.
18. J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1, 2010.
19. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.
20. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.
21. S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB*, pages 386–395, 1997.
22. S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDBJ*, 12(3):244 – 261, 2003.
23. A. Ibrahim and G. H. L. Fletcher. Efficient processing of containment queries on nested sets. In *EDBT*, pages 227–238, 2013.
24. R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *DASFAA*, pages 761–772, 2005.
25. Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.
26. H. Köhler. Estimating set intersection using small samples. In *ACSC*, pages 71–78, 2010.
27. N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD Conference*, pages 157–168, 2003.
28. S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *EDBT*, pages 427–444, 2002.
29. S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28:56–99, 2003.
30. K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, 2000.
31. R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *DMKD*, pages 20–27, 2003.
32. R. Rantzau, L. D. Shapiro, B. Mitschang, and Q. Wang. Algorithms and applications for universal quantification in relational databases. *Inf. Syst.*, 28(1-2):3–32, 2003.
33. L. Ribeiro and T. Härder. Efficient set similarity joins using min-prefixes. In *Advances in Databases and Information Systems, 13th East European Conference, ADBIS 2009, Riga, Latvia, September 7-10, 2009. Proceedings*, pages 88–102, 2009.
34. S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.
35. S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.
36. S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *SIGIR*, pages 738–739, 2009.
37. M. Terrovitis, P. Bouros, P. Vassiliadis, T. K. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *EDBT*, pages 225–236, 2011.

38. M. Terrovitis, S. Passas, P. Vassiliadis, and T. K. Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737, 2006.
39. D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *PVLDB*, 2(1):838–849, 2009.
40. J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.
41. C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.
42. X. Zhang, K. Chen, L. Shou, G. Chen, Y. Gao, and K.-L. Tan. Efficient processing of probabilistic set-containment queries on uncertain set-valued data. *Inf. Sci.*, 196:97–117, 2012.
43. Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *KDD*, pages 401–406, 2001.
44. J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *TOIS*, 23(4):453–490, 1998.